ENCS5337: Chip Design Verification

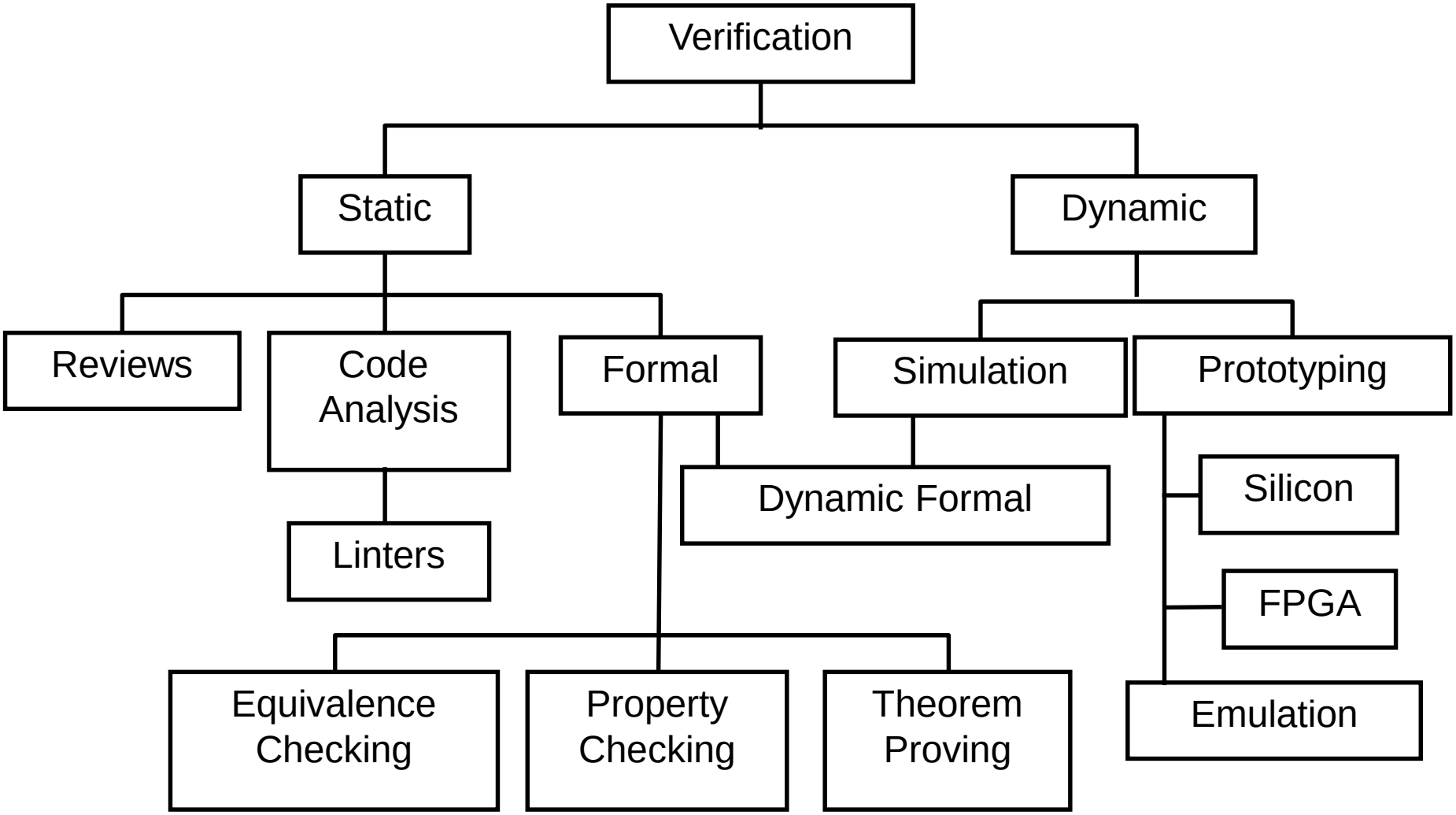
Spring 2023/2024

Verification Tools

Dr. Ayman Hroub

Many thanks to Dr. Kerstin Eder for most of the slides

Functional Verification Approaches



Achieving Automation

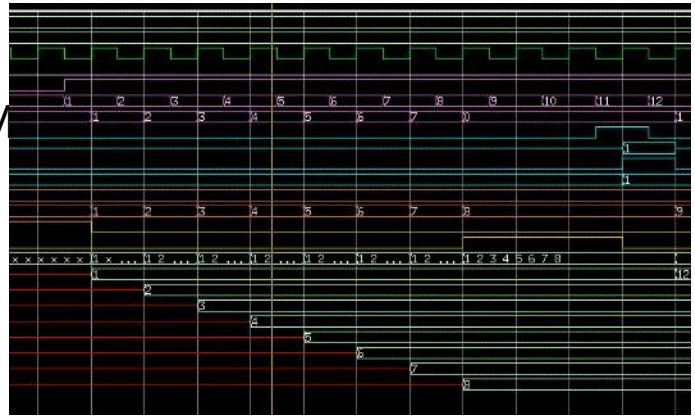
Task of Verification Engineer:

- Ensure product does not contain bugs - as fast and as cost-effective as possible.
- Parallelism, Abstraction and **Automation** can reduce the duration of verification.
- Automation reduces human factor, improves efficiency and reliability.

Tools used for Verification

- Dynamic Verification:
  - Hardware Verification Languages (HVL)
  - Testbench automation
  - Test generators
  - Coverage collection and analysis
  - General purpose HDL Simulators
    - Event-driven simulation
    - Cycle-based simulation (improved performance)
    - Waveform viewers (for debug)
  - Hardware accelerators/emulators, FPGAs

- Static Analysis / Verification Methods (Formal M
  - Linting Tools
  - Equivalence checkers
  - Model checkers
    - Property Specification Languages (ABV)
  - Theorem provers



Some Commercial EDA DV Tools

Company	Simulation based EDA Tools	Formal Verification EDA Tools
Synopsys	VCS	Formality, VCS Formal
Cadence	Xcelium	JasperGold
Siemens	Questa	FormalPro

Linting Tools

- Linters are **static** checkers.
- Assist in finding common coding mistakes
  - Linters exist for software and also for hardware.
    - `gcc -Wall` (When do you use this?)
- Only identify certain classes of problems
  - Many false positives are reported.
  - Use a **filter** program to reduce these.
    - Careful - don't filter true positives though!
- Does assist in enforcing **coding guidelines!**
- Rules for coding guidelines can be added to linter.

Simulation-Based Verification

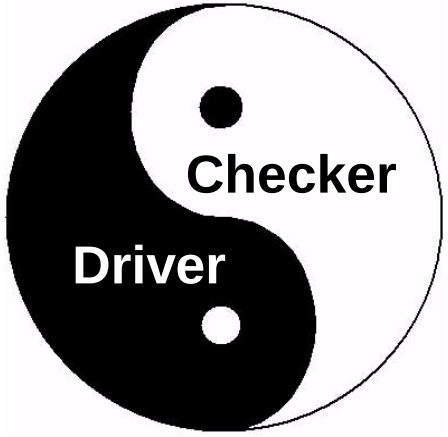
Directed testing with manual checking

Fundamentals of Simulation-based Verification

- Verification can be divided into two separate tasks
  1. Driving the design - Controllability
  2. Checking its behavior - Observability
- Basic questions a verification engineer must ask
  1. Am I driving *all possible* input scenarios?
  2. How will I know when a failure has occurred?

How do I know when I'm done?

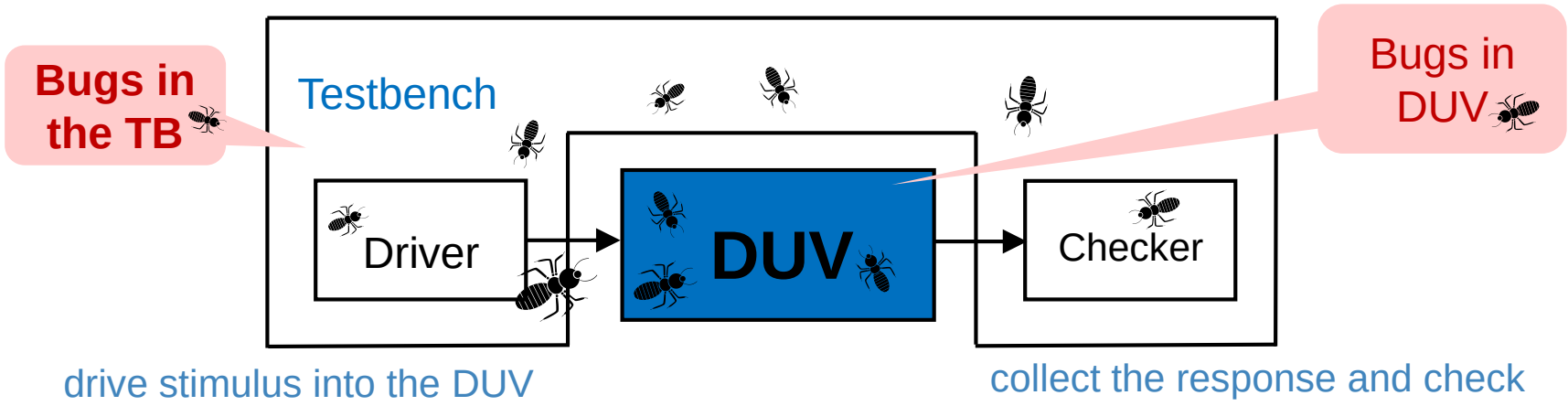
- Driving and checking are the yin and yang of verification
  - We cannot find bugs without creating the failing conditions
    - Drivers
  - We cannot find bugs without detecting the incorrect behavior
    - Checkers



What is a Testbench?

*"**Code** used to create a predetermined **input sequence** to a design, and to then observe the **response**."*

- Generic term used differently across the industry.
- Always refers to a test case/scenario.
- Traditionally, a testbench refers to code **written in a Hardware Description Language (VHDL, Verilog, SystemVerilog) at the top level of the design hierarchy**.
- **A testbench is a "completely closed" system, i.e.c, no input/output of the testbench**



Simulation-based Design Verification

- Simulate the design (not the implementation) **before** fabrication.
- **Simulating the design relies on simplifications:**
  - Functional correctness/accuracy can be a problem.

Verification Challenge: *"What input patterns to supply to the Design Under Verification (DUV) ..."*

- Simulation requires **stimulus.** It is dynamic, not just static!
- Requires to reproduce environment in which design will be used.
  - **Testbench** (Remember: Verification vs Testing!)

Verification Challenge: *"... and knowing what is expected at the output for a properly working design."*

- **Simulation outputs are checked externally** against design intent (specification)
  - Errors cannot be proven not to exist!

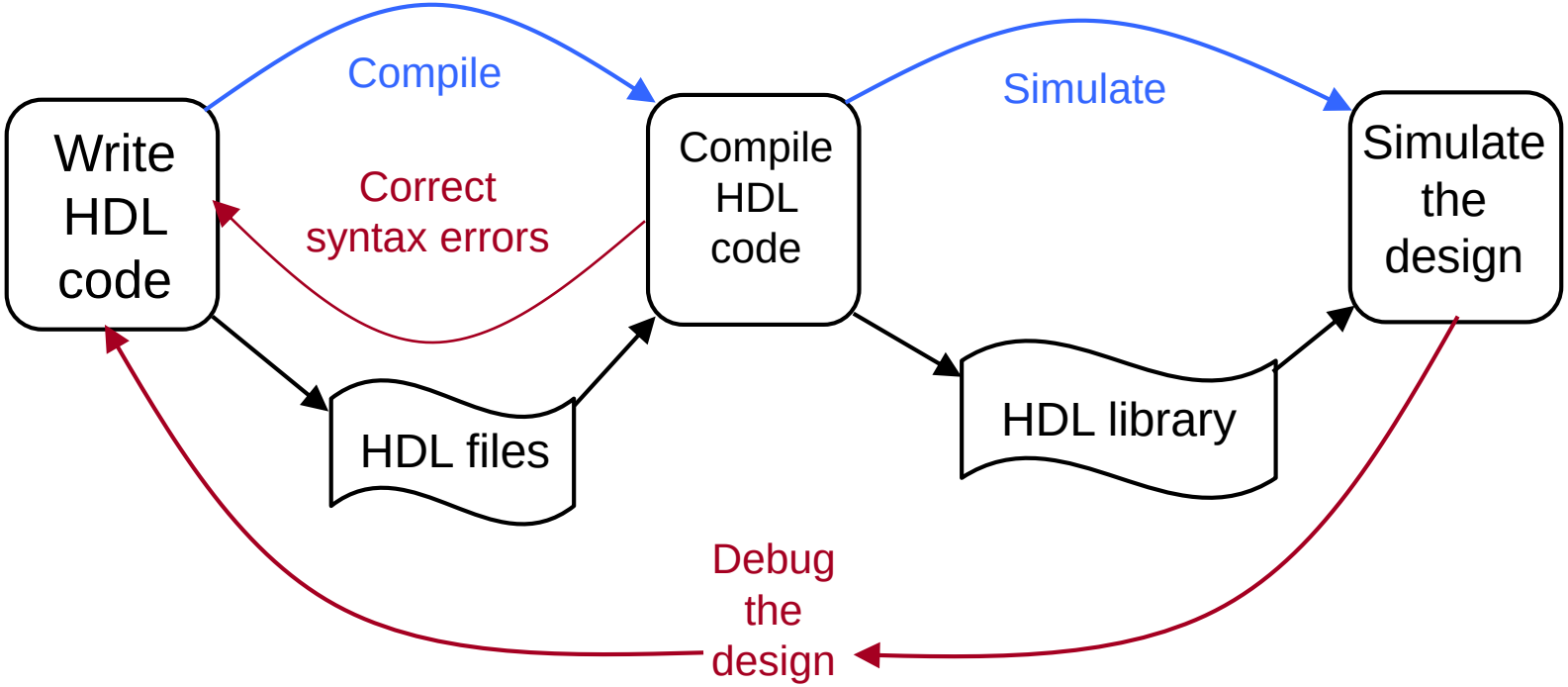
"Testing shows the presence, not the absence of bugs."

[Edsger W. Dijkstra]

Two types of simulators: **event-based** and **cycle-based**

Simulation based on Compiled Code

- To simulate with **ModelSim:**
  - Compile HDL source code into a library.
  - Compiled design can be simulated.



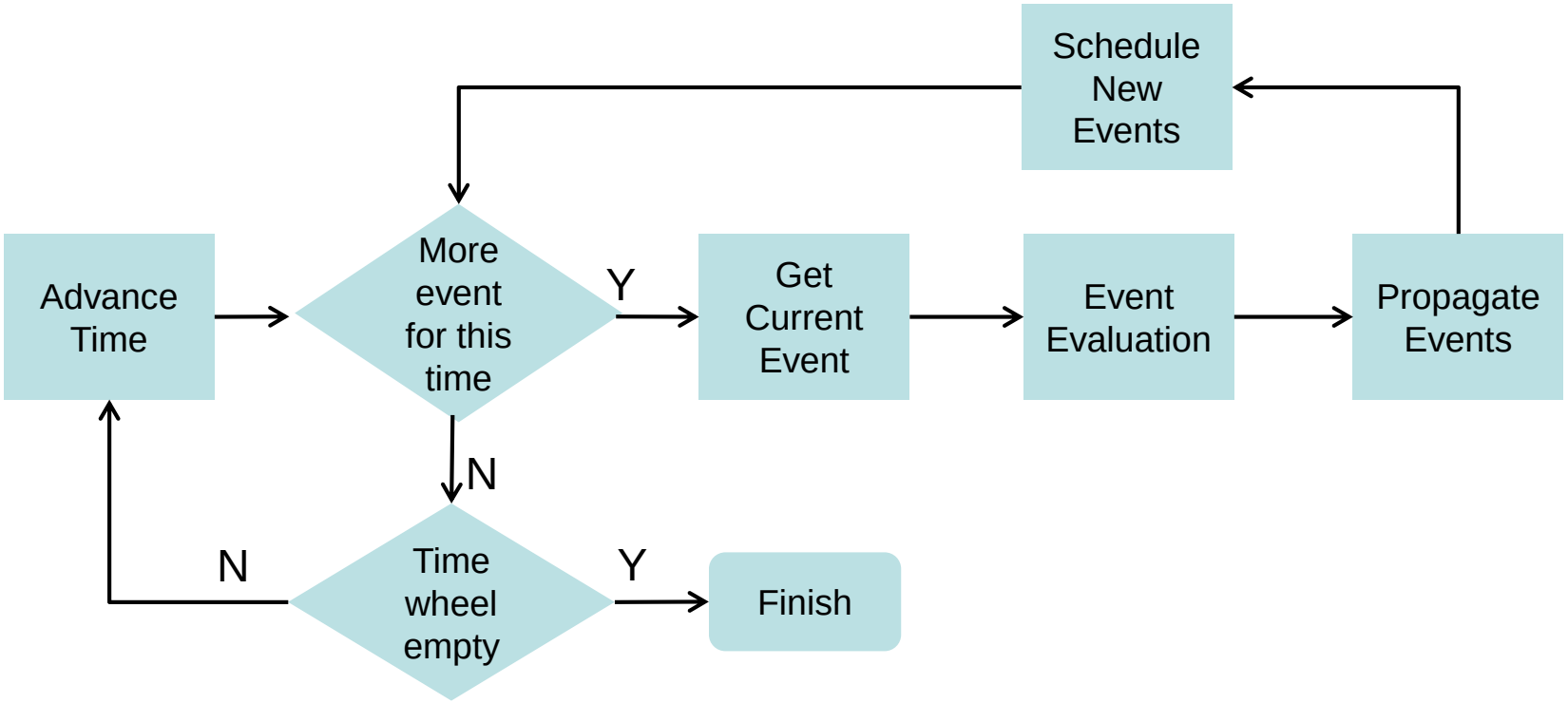
Event-based Simulators (I)

Event-based simulators are driven based on **events.**

- Outputs are a function of inputs:
  - The outputs change only when the inputs do.
  - **The event is the input changing.**
  - An event causes the simulator to re-evaluate and calculate new output.
- Outputs (of one block) may be used as inputs (of another) ...

Event-Driven Simulators (2)

- Accurate timing
- Good debugging environment
- Simulation speed is slower



Event Driven Principles

- The event simulator maintains many lists:
  - A list of all **atomic** executable blocks
  - Fanout lists: A data structure that represents the interconnect of the blocks via signals
  - **A time queue** – points in time when events happen
  - **Event queues** – one queue pair for each entry in the time queue
    - Signal update queue
    - Computation queue
- The simulator needs to process all these queues at simulation time.

Event-based Simulators

Event-based simulators are driven based on **events.** ☺

- Outputs are a function of inputs:
  - The outputs change only when the inputs do.
  - **The event is the input changing.**
  - An event causes the simulator to re-evaluate and calculate new output.
- Outputs (of one block) may be used as inputs (of another) ...
- **Re-evaluation happens until no more changes propagate through the design.**

Simulation Speed

What is holding us back?

Speedup strategies

Improving Simulation Speed

- The most obvious bottle-neck for functional verification is simulation throughput

- There are several ways to improve throughput
  - Parallelization
  - Compiler optimization techniques
  - Changing the level of abstraction
  - Methodology-based subsets of HDL
    - Cycle-based simulation
  - Special simulation machines

Parallelization

- Efficient parallel simulation algorithms are hard to develop
  - Much parallel event-driven simulation research
  - Has not yielded a breakthrough
  - Hard to compete against "trivial parallelization"
- Simple solution – run independent testcases on separate machines

Principle of Operation

- Compiler transforms combinational logic into Boolean operations
- Compiler schedules inter-processor communications using a fast broadcast technique
- Performance dictated by
  - Number of processors
  - Number of levels in the design

Simulation Speed Comparison

Event Simulator	1
Cycle Simulator	20
Event driven cycle Simulator	50
Acceleration	1000
Emulation	100000

Verification Languages

Raising the level of abstraction

Verification Languages

- Need to be designed to address **verification** principles.
- Deficiencies in RTL languages (HDLs such as Verilog and VHDL):
  - **Verilog** was designed with focus on describing low-level hardware structures.
    - No support for **data structures** (records, linked lists, etc).
    - Not object/aspect-oriented.
- Limitations inhibit **efficient** implementation of verification strategy.
- High-level verification languages are (currently):
  - **System Verilog**
  - e-language used for Cadence's Specman Elite [IEEE P1647]
  - (Synopsys' Vera, SystemC)

Features of High-Level Verification Languages

- Raising the level of abstraction:
  - From bits/vectors to high-level data types/structures
    - lists, structs, scoreboards including ready made functions to access these
- Support for building the verification environment
  - Enable testbench automation
  - Modularity
    - Object/aspect oriented languages
    - Libraries (VIP) to enable re-use
- Support for test generation
  - Constrained random test generation features
    - Control over randomization to achieve the target values
    - Advanced: Connection to DUV to generate stimulus depending on DUV state
- Support for coverage
  - Language constructs to implement functional coverage models

Directed Testing

Focus on checking

The Importance of Driving and **Checking**

- Drivers activate the bug.
- The observable effects of the bug then need to propagate to a checker.
- A checker needs to be in place to detect the incorrect behaviour.

All three are needed to find bugs!

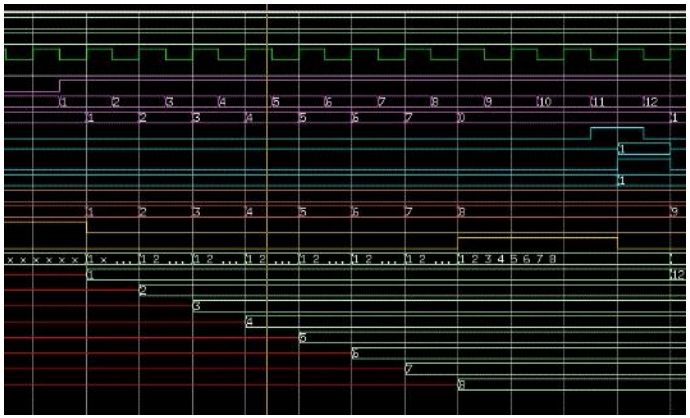
Checking: How to predict expected results

- Methods for checking:
  - **Directed testing:**
    - Because we know what will be driven, a checker can be developed for each test case individually.

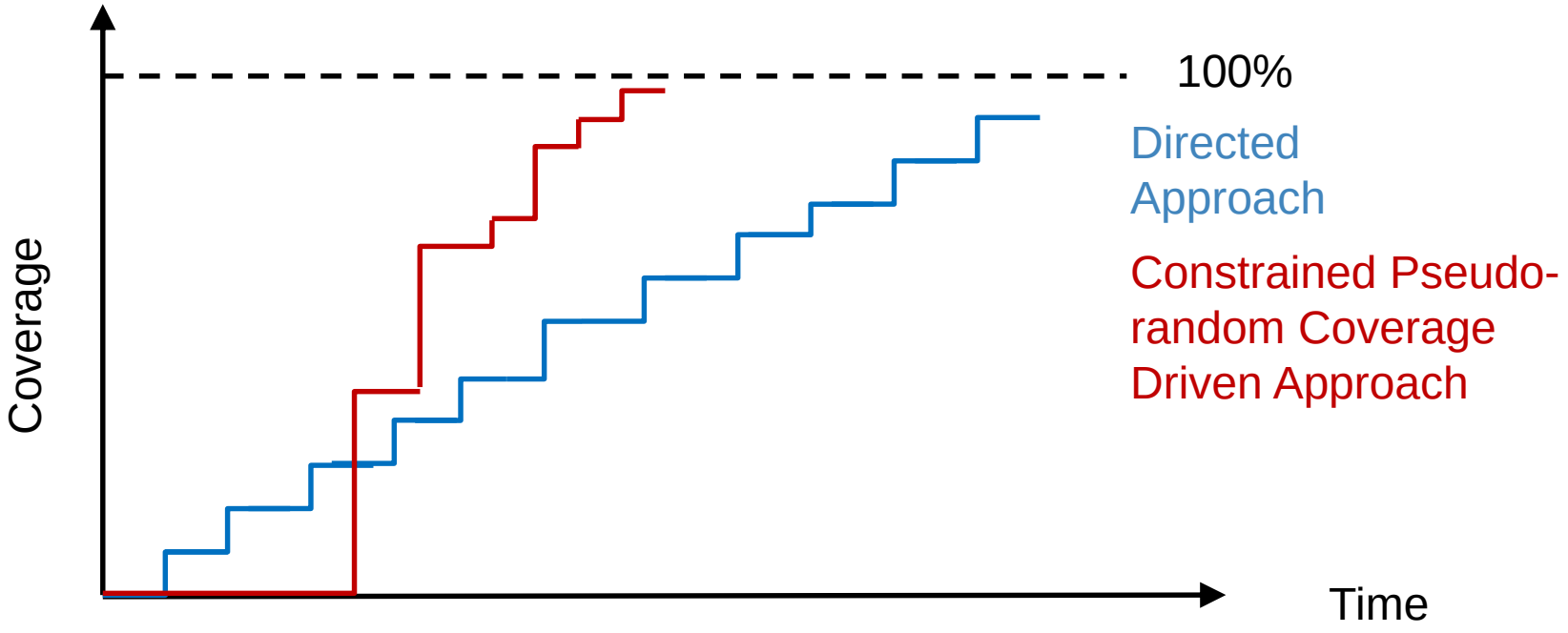
- Sources for checking:
  - Understanding of the inputs, outputs and the transfer function of the DUV.
  - Understanding of the design context.
  - Understanding of the internal structures and algorithms (uarch).
  - Understanding of the top-level design description (arch).
  - **Understanding of the specification.**

Limitations of Using Waveform Viewers as Checkers

- Often come as part of a simulator.
- **Most common verification tools used...**
  - Used to **visually inspect** design/testbench/verification environment.
  - Recording waves decreases performance of simulator. (Why?)
- **Don't use viewer to determine if DUV passes/fails a test.**
  - **Why not?**
- **Can use waveform viewer for debugging.**
  - Consider costs and alternatives.
  - Benefits of automation.
  - Need to increase productivity.



Limitations of Directed Testing: Coverage



Criteria:

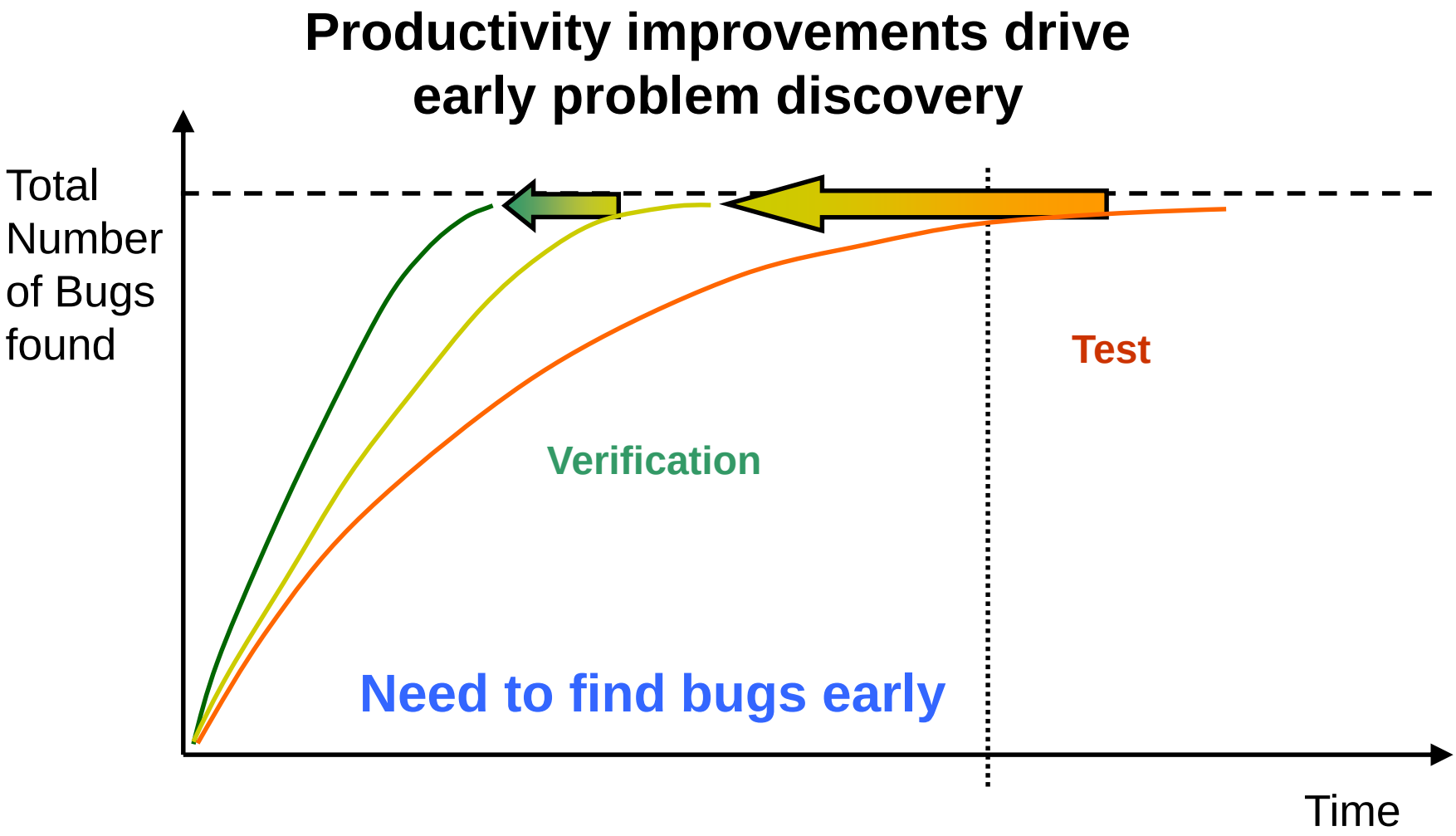
- Effectiveness
- Efficiency
- Maintainability
- Re-usability

Directed testing has many shortfalls wrt these criteria.

Why would one use Directed Testing?

Need to increase productivity!

Impact of Increasing Verification Productivity



Verification Tools

Third Party Models

Metrics

Third Party Models

- Chip needs to be verified in its **target environment.**
  - Board/SoC Verification
- Do you develop or purchase behavioural models (specs) for board parts?
  - Buying them may seem expensive!

  "If it was not worth designing on your own to begin with, why is writing your own model now justified?"
  - The model you develop is not as reliable as the one you buy.
  - The one you buy is used by many others - not just yourself.
- Remember: In practice, it is often more expensive to develop your own model to the **same degree of confidence** than licensing one.

Metrics

- Not really verification tools - but managers love metrics and measurements!
  - Managers often have little time to personally assess progress.
  - They want something measurable.
- **Coverage** is one metric - will be introduced later.
- **Others metrics include:**
  - Number of lines of code
  - Ratio of lines of code (between design and verifier)
  - Drop of source code changes
  - Number of outstanding issues

Summary

We have covered:

- Verification Tools & Languages
- Basic testbench components
- Writing directed tests
- The importance of Driving and Checking
- Checking when we use directed testing
- Limitations of directed testing
- Cost of debug using waveforms